CAPÍTULO 6

THREAD

Prezado(a) aluno(a),

Neste capítulo você aprenderá o que são threads e as vantagens de sua utilização em relação a processos pesados. Verá os ambientes monothreads e multithreads bem como seus diferentes tipos de arquitetura e implementação.

Bom estudo!

Fala Professor

6.1. Introdução

Até o final da década de 1970, os sistemas operacionais suportavam apenas processos com um único *thread* (monothread), ou seja, um processo com apenas um único programa fazendo parte do seu contexto. A partir do conceito de múltiplos threads (multithread), foi possível projetar e implementar aplicações concorrentes de forma eficiente, pois um processo pôde ter partes diferentes do seu código sendo executadas em paralelo com um *overhead* (sobrecarga) menor que ao utilizar múltiplos processos.

Como as *threads* de um mesmo processo compartilham o mesmo espaço de endereçamento, a comunicação entre *threads* não envolve mecanismos lentos de intercomunicação entre processos, aumentando, consequentemente, o desempenho da aplicação [1, 2, 3, 4].

O desenvolvimento de programas que exploram os benefícios da programação multithread não é simples. A presença do paralelismo introduz um novo conjunto de problemas como a comunicação e sincronização de threads. Existem diferentes modelos para a implementação de threads em um sistema operacional, onde desempenho, flexibilidade e custo devem ser avaliados.

Fala Professor

Atualmente, o conceito de multithread pode ser encontrado em diversos sistemas operacionais, como no GNU/Linux, Sun Solaris e Microsoft Windows 2000/XP. A utilização comercial de sistemas operacionais multithreads é crescente em função do aumento da popularidade dos sistemas com múltiplos processadores, do modelo cliente-servidor e dos sistemas distribuídos.

6.2. Ambiente monothread

Um programa é uma sequência de instruções, composta por desvios, repetições e chamadas a procedimentos e funções. Quando um programa está sendo executado, há um registrador – chamado PC (*Program Counter*) ou IP (*Instruction Pointer*) – que aponta para o endereço da próxima instrução a ser executada. Ao longo da execução de um programa, os endereços de instruções apontados por esse registrador formam um caminho denominado "**fluxo de execução**" ou "**fluxo de controle**" (alguns também chamam de "**linha de execução**").

Conceitos

Thread pode ser definido como um "**fluxo de execução**" (ou "fluxo de controle") dentro de um programa.

Obs: É importante lembrar que todo processo possui, no mínimo, uma *thread* (ou fluxo de execução).

Em um ambiente monothread, um processo suporta apenas uma única *thread*. Nesse ambiente, cada processo possui seu próprio contexto de hardware, contexto de software e espaço de endereçamento. Além disso, aplicações concorrentes são implementadas apenas com o uso de múltiplos processos independentes ou subprocessos. A utilização de processos independentes e subprocessos permitem dividir uma aplicação em partes que podem trabalhar de forma concorrente.

Fala Professor

Um exemplo do uso de concorrência pode ser encontrado nas aplicações com interface gráfica, como em um software de gerenciamento de e-mails. Nesse ambiente, um usuário pode ler suas mensagens antigas, ao mesmo tempo em que envia e-mails e recebe novas mensagens.

Com o uso de múltiplos processos, cada funcionalidade do software implicaria na criação de um novo processo pra atendê-la, aumentando o desempenho da aplicação.

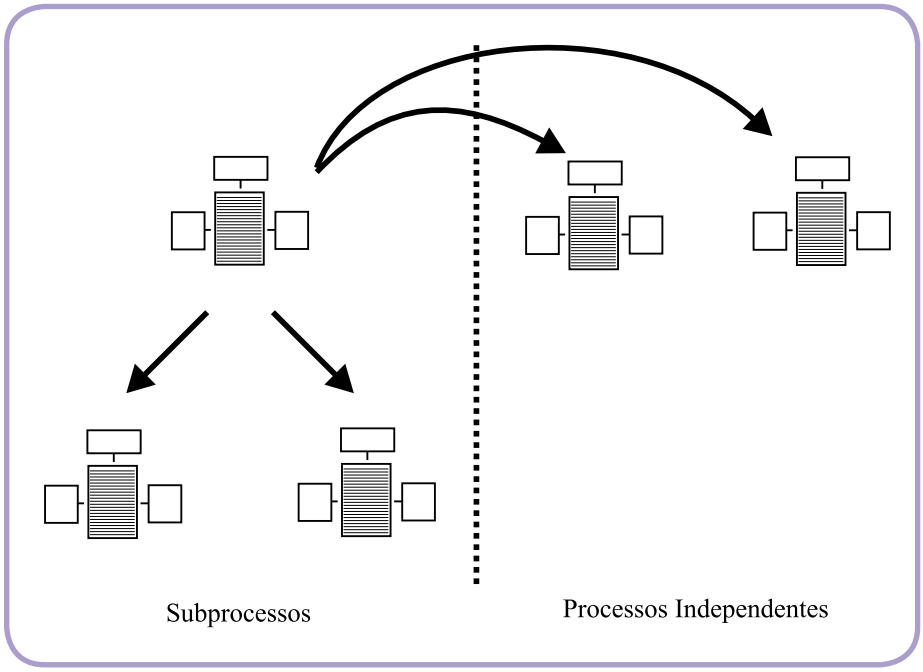

Figura 6-42: Concorrência com subprocessos e processos independentes
Fonte: [1] – Machado e Maia, 2004. Adaptação.

O problema nesse tipo de implementação é que o uso de processos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos para cada processo, consumindo tempo de processador nesse trabalho. No caso do término do processo, o sistema dispensa tempo para desalocar recursos previamente alocados.

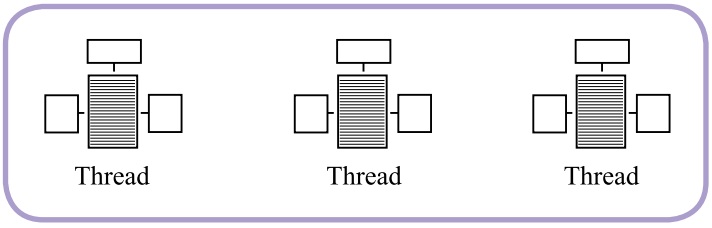

Figura 6-43: Ambiente monothread
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Outro problema a ser considerado é quanto ao compartilhamento do espaço de endereçamento. Como cada processo possui seu próprio espaço de endereçamento, a comunicação entre processos torna-se difícil e lenta, pois utiliza mecanismos como *pipes*, sinais, semáforos, memória compartilhada ou troca de mensagem. Além disso, o compartilhamento de recursos comuns aos processos concorrentes, como memória e arquivos abertos, não é simples.

6.3. Ambiente multithread

Em um ambiente multithread, cada processo pode possuir vários fluxos de execução – ou *threads*. Nesse ambiente, um único processo, que tem pelo menos uma thread, pode possuir várias threads de execução que compartilham o mesmo espaço de endereçamento, o mesmo contexto de software e possuem seus próprios contextos de hardware e pilha [1].

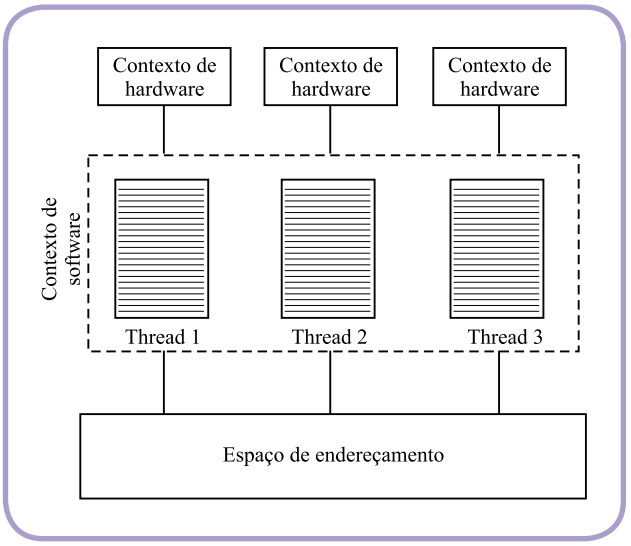

Figura 6-44: Ambiente multithread
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Para que as threads sejam criadas, é necessário que sejam definidos no programa principal. De forma simplificada, uma thread pode ser definida como uma sub-rotina de um programa que pode ser executada de forma assíncrona, ou seja, executada concorrentemente ao programa chamador. O programador deve especificar as *threads*, associando-as às sub-rotinas assíncronas. Dessa forma, um ambiente multithread possibilita a execução concorrente de sub-rotinas dentro de um mesmo processo. Por exemplo, suponha que exista um programa principal que realize a chamada de duas sob-rotinas assíncronas – através de *threads*. Inicialmente, o processo é criado apenas com uma *thread*, para a execução do programa principal. Quando o programa principal chamar as sub-rotinas, serão criados as duas *threads,* e estas, serão executadas independentes da *thread* do programa principal. Neste processo, as três *threads* serão executadas concorrentemente.

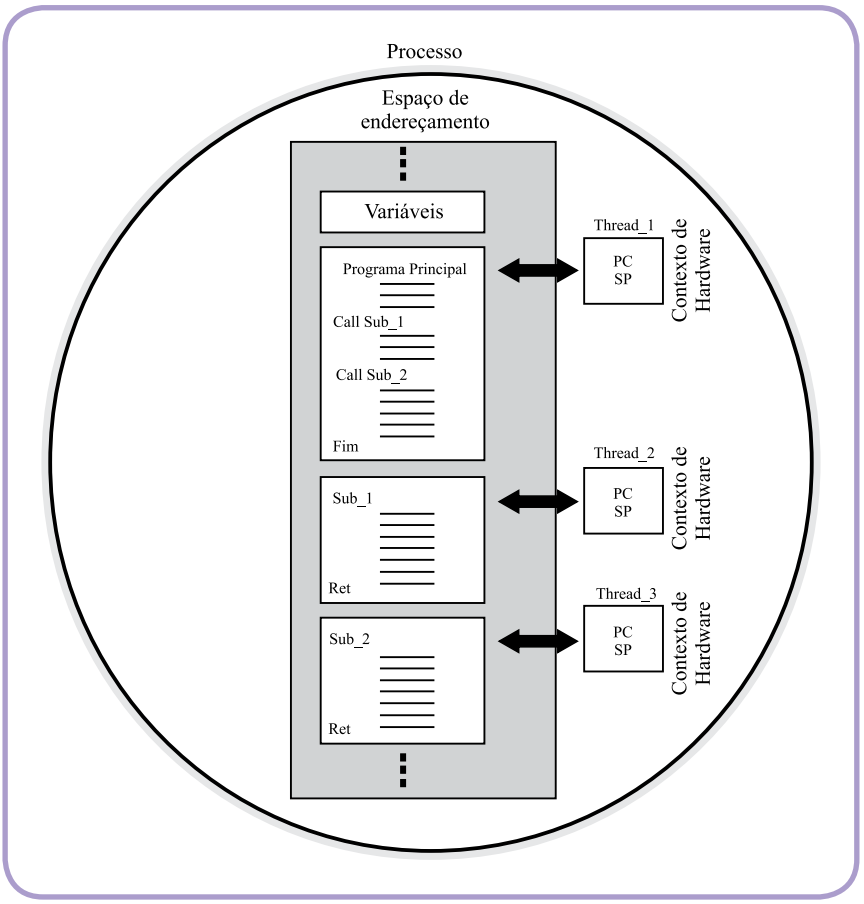

Figura 6-45: Aplicação multithread
Fonte: [1] – Machado e Maia, 2004. Adaptação.

No ambiente multithread, cada processo pode responder a várias solicitações concorrentemente ou mesmo simultaneamente, caso haja mais de um processador. A grande vantagem no uso de *threads* é a possibilidade de minimizar a alocação de recursos do sistema, além de diminuir o *overhead* na criação, troca e eliminação de processos.

Threads compartilham o processador da mesma maneira que processos e passam pelas mesmas mudanças de estado (execução, espera e pronto). Por exemplo, enquanto uma thread espera por uma operação de E/S, outra thread pode ser executada. Para permitir a troca de contexto entre diversas threads, cada thread possui seu próprio contexto de hardware, com o conteúdo dos registradores gerais e específicos. Quando uma thread está sendo executada, seu contexto de hardware está armazenado nos registradores do processador. No momento em que a thread perde a utilização da CPU, as informações são atualizadas no seu contexto de hardware.

Dentro de um mesmo processo, *threads* compartilham o mesmo contexto de software e espaço de endereçamento com as demais *threads*, porém, cada *thread* possui seu contexto de hardware individual e sua pilha individual. Threads são implementados internamente por meio de uma

estrutura de dados denominada **bloco de controle da thread** (*Thread Control Block* – TCB). O TCB armazena, além do contexto de hardware, mais algumas informações relacionadas exclusivamente a *thread*, como prioridade, estado de execução e bits de estado.

Em ambientes monothread, o processo é, ao mesmo tempo, a unidade de alocação de recursos e a unidade de escalonamento. A independência entre os conceitos de processo e *thread* permite separar a unidade de alocação de recursos da unidade de escalonamento, que em ambientes monothread estão fortemente relacionadas. Em um ambiente multithread, a unidade de alocação de recursos é o processo, onde todas as suas *threads* compartilham o espaço de endereçamento, descritores de arquivos e dispositivos de E/S. Cada *thread* representa uma unidade de escalonamento independente. Nesse caso, o sistema não seleciona um processo para a execução, mas sim, uma de suas *threads*.

A grande diferença entre aplicações monothread e multithread está no uso do espaço de endereçamento. Processos independentes e subprocessos possuem espaços de endereçamento individuais e protegidos, enquanto threads compartilham o espaço dentro de um mesmo processo. Essa característica permite que o compartilhamento de dados entre *threads* de um mesmo processo seja mais simples e rápido, se comparado a ambientes monothread.

Como *threads* de um mesmo processo compartilham o mesmo espaço de endereçamento, não existe qualquer proteção no acesso à memória, o que permite que uma *thread* possa alterar facilmente dados de outros. Para que *threads* trabalhem de forma cooperativa, é fundamental que a aplicação implemente mecanismos de comunicação e sincronização entre elas, a fim de garantir o acesso seguro aos dados compartilhados na memória.

O uso de multithreads proporciona uma série de benefícios. Programas concorrentes com múltiplas *threads* são mais rápidos do que programas concorrentes implementados com múltiplos processos, pois operações de criação, chaveamento de contexto e eliminação das *threads* geram menor *overhead*. Como as *threads,* dentro de um processo, dividem o mesmo espaço de endereçamento, a comunicação entre elas pode ser realizada de forma rápida e eficiente. Além disso, *threads* em um mesmo processo podem compartilhar facilmente outros recursos, como descritores de: arquivos, temporizadores, sinais, atributos de segurança, etc.

A utilização do processador, dos discos e de outros periféricos pode ser feita de forma concorrente pelas diversas *threads*, significando melhor utilização dos recursos computacionais disponíveis. Em algumas apli-

cações, a utilização de *threads* pode melhorar o desempenho da aplicação apenas executando tarefas em background enquanto operações E/S estão sendo processadas. Aplicações como: editores de texto, planilhas, aplicativos gráficos e processadores de imagens, são especialmente beneficiados quando desenvolvidos com base em *threads*.

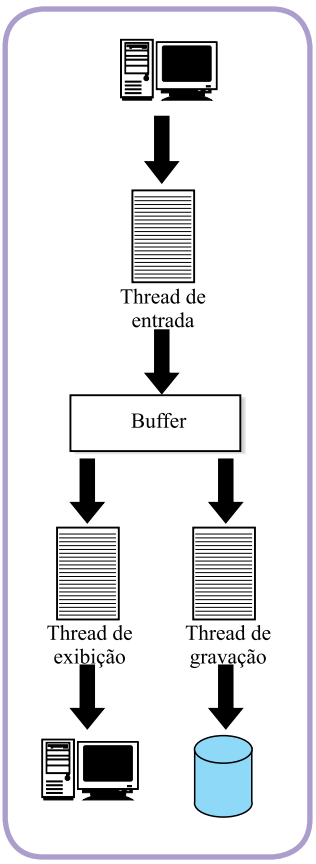

Figura 6-46: Exemplo de aplicação multithread (entrada/saída)
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Em ambientes cliente-servidor, *threads* são essenciais para solicitações de serviços remotos. Em um ambiente monothread, se uma aplicação solicita um serviço remoto, ela pode ficar esperando indefinidamente, enquanto aguarda pelo resultado. Em um ambiente multithread, uma *thread* pode solicitar o serviço remoto, enquanto a aplicação pode continuar realizando outras atividades. Já para o processo que atende a solicitação, múltiplas *threads* permitem que diversos pedidos sejam atendidos simultaneamente.

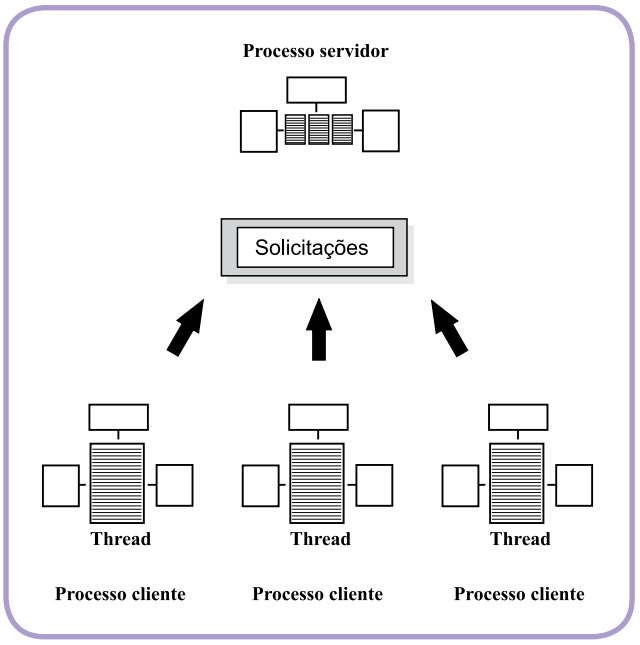

Figura 6-47: Exemplo de aplicação multithread (cliente-servidor)
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Não apenas aplicações tradicionais podem fazer uso dos benefícios do multithreading; o núcleo do sistema operacional também pode ser implementado com o uso dessa técnica de forma vantajosa, como na arquitetura microkernel, apresentada anteriormente.

Atividades

Atividades

1. Como uma aplicação pode implementar concorrência em um ambiente monothread?
2. Quais os problemas de aplicações concorrentes desenvolvidas em ambientes monothread?
3. O que é um *thread* e quais as vantagens em sua utilização?
4. Explique a diferença entre unidade de alocação de recursos e unidade de escalonamento?
5. Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre *threads* de um mesmo processo?

6.4. Arquitetura e implementação

O conjunto de rotinas disponíveis para que uma aplicação utilize as facilidades das *threads* é chamado de "pacote de *threads*". Existem diferentes abordagens na implementação desse pacote em um sistema operacional, o que influenciará no desempenho, na concorrência e na modularidade das aplicações multithread [1].

Threads podem ser oferecidas por uma biblioteca de rotinas fora do núcleo do sistema operacional (**modo usuário**), pelo próprio núcleo do sistema (**modo kernel**), por uma combinação de ambos (**modo híbrido**) ou por um modelo conhecido como ***Scheduler Activations***.

Uma das grandes dificuldades para a utilização de *threads* foi à ausência de um padrão. Em 1995, o padrão POSIX P1003.1c foi aprovado e, posteriormente, atualizado para a versão POSIX 1003.4a. Com esse padrão, também conhecido como **Pthreads**, aplicações comerciais multithreading tornaram-se mais simples e de fácil implementação. O padrão Pthreads é largamente utilizado em ambientes Unix, como o Sun Solaris Pthreads e o DECthreads para Digital OSF/1.

6.4.1. Threads em modo usuário

Threads em modo usuário (**TMU**) são implementadas pela aplicação e não pelo sistema operacional. Para isso, deve existir uma biblioteca de rotinas que possibilite à aplicação "realizar tarefas", como "criação/eliminação de *threads*", "troca de mensagens entre threads" e uma política de escalonamento. Nesse modo, o sistema operacional não sabe da existência de múltiplas *threads*, sendo responsabilidade exclusiva da aplicação, gerenciar e sincronizar as diversas *threads* existentes.

A vantagem desse modelo é a possibilidade de implementar aplicações multithreads mesmo em sistemas operacionais que não suportam *threads*. Utilizando a biblioteca, múltiplos *threads* podem ser criados, compartilhando o mesmo espaço de endereçamento do processo, além de outros recursos. TMU são rápidos e eficientes por dispensarem acessos ao kernel do sistema operacional, evitando assim, a mudança de modo de acesso (usuário-kernel-usuário).

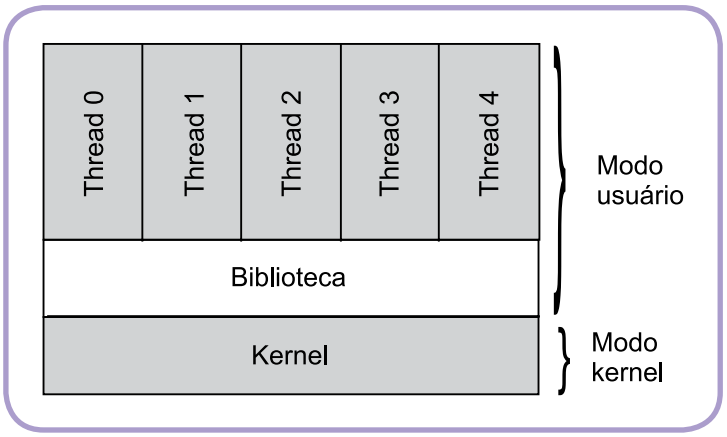

Figura 6-48: Threads em modo usuário
Fonte: [1] – Machado e Maia, 2004. Adaptação.

As TMU possuem uma grande limitação, pois o sistema operacional gerencia cada processo como se existisse somente uma única *thread*. No momento em que uma chama uma rotina do sistema que o coloca em estado de bloqueado/espera (rotina bloqueante) todo o processo é colocado no estado de bloqueado/espera, mesmo havendo outras *threads* prontas para execução.

Outra limitação das TMU ocorre em ambientes com múltiplos processadores. Como reconhece apenas processos e não *threads*, o sistema seleciona apenas processos para execução e não *threads*. Com isso, não é possível que múltiplas *threads* de um processo possam ser executados em diferentes CPUs simultaneamente. Essa restrição limita drasticamente o grau de paralelismo da aplicação, já que as *threads* de um mesmo processo, podem ser executadas em somente um processador, de cada vez.

6.4.2. Threads em modo kernel

Threads em modo kernel (**TMK**) são implementadas diretamente pelo núcleo do sistema operacional, por meio de chamadas a rotinas do sistema que oferecem todas as funções de gerenciamento e sincronização. O sistema operacional sabe da existência de cada t*hread* e pode escalonálas individualmente. No caso de múltiplos processadores, as *threads* de um mesmo processo podem ser executadas simultaneamente, aumentando com isso, o paralelismo da aplicação.

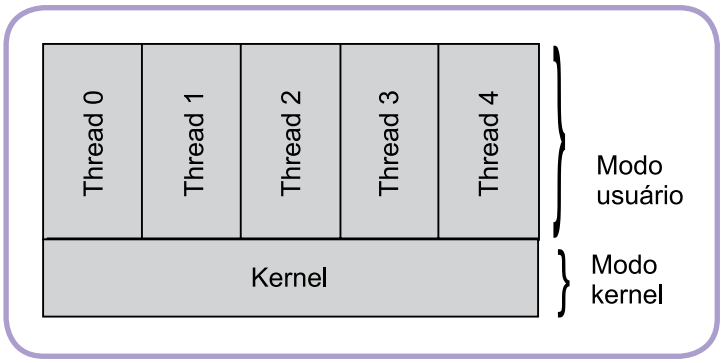

Figura 6-49: Threads em modo kernel
Fonte: [1] – Machado e Maia, 2004. Adaptação.

A desvantagem para pacotes em modo kernel é possuírem menor desempenho em seu tratamento, se comparados com pacotes em modo usuário. Isso ocorre porque, nos pacotes em modo usuário, todo tratamento é feito sem a ajuda do sistema operacional, ou seja, sem a mudança do modo de acesso (usuário-kernel-usuário); ao passo que nos pacotes em modo kernel são utilizadas chamadas às rotinas do sistema e, consequentemente, várias mudanças no modo de acesso.

6.4.3. Threads em modo híbrido

A arquitetura de **threads em modo híbrido** procura combinar as vantagens de *threads*, implementadas em modo usuário (TMU) e threads em modo kernel (TMK). Um processo pode ter vários TMK e, por sua vez, um TMK pode ter vários TMU. O núcleo do sistema reconhece os TMK e pode escaloná-los individualmente. Um TMU pode ser executado em um TMK, em um determinado momento, e no instante seguinte ser executado em outro.

O programador desenvolve a aplicação em termos de TMU e especifica quantos TMK estão associados ao processo. Os TMU são mapeados em TMK enquanto o processo está sendo executado. O programador pode utilizar apenas TMK, TMU ou uma combinação de ambos.

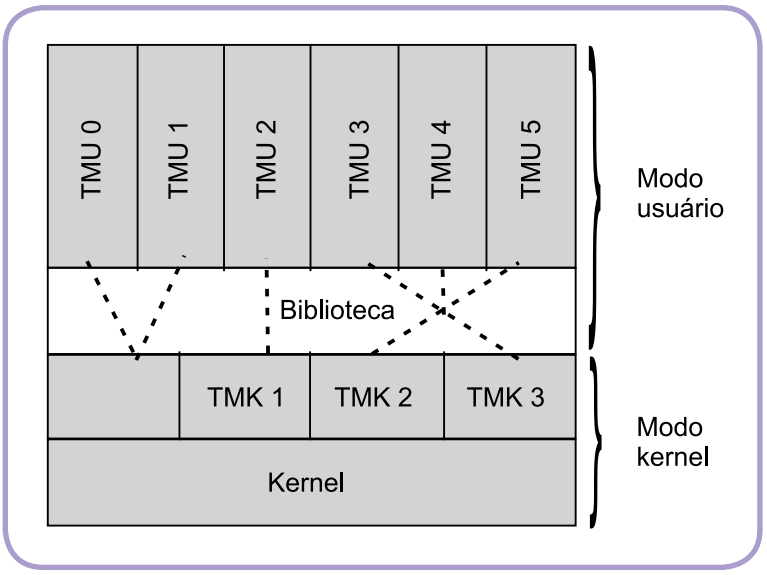

Figura 6-50: Threads em modo híbrido
Fonte: [1] – Machado e Maia, 2004. Adaptação.

O modo híbrido, apesar da maior flexibilidade, apresenta problemas herdados de ambas as implementações. Por exemplo, quando um TMK realiza uma chamada bloqueante, todos os TMU são colocados no estado de espera. TMU que desejam utilizar vários processadores devem utilizar diferentes TMK, o que influenciará no desempenho. A grande dificuldade está em definir quantos TMU devem ser colocados em cada TMK. Em geral, procura-se colocar as TMU do tipo IO-bound em TMK separadas, para que seu bloqueio não afete outras TMU.

6.4.4. Scheduler activations

Os problemas apresentados no pacote de *threads* em modo híbrido existem devido à falta de comunicação entre as *threads* em modo usuário e em modo kernel. O modelo ideal deveria utilizar as facilidades do pacote em modo kernel, com o desempenho e flexibilidade do modo usuário.

O modelo *Scheduler Activations* foi criado no início da década de 1990, na Universidade de Washington. Esse modelo combina o melhor das duas arquiteturas, mas em vez de dividir as *threads* em modo usuário entre os de modo kernel, o núcleo do sistema troca informações com a biblioteca de *threads* utilizando uma estrutura de dados chamada ***Scheduler Activations***.

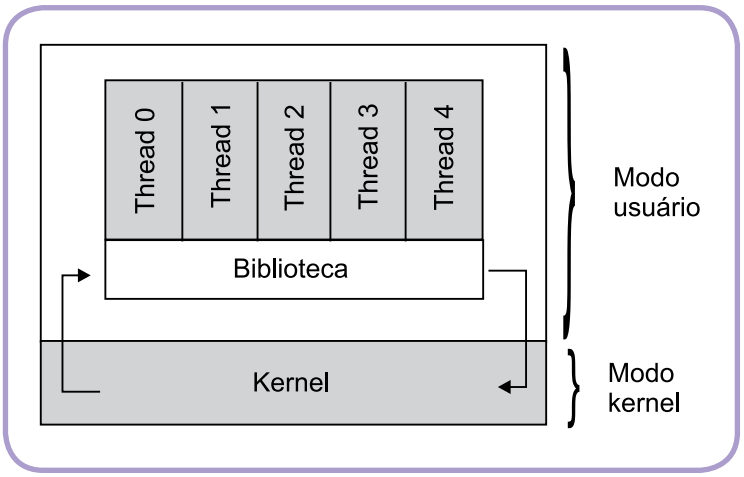

Figura 6-51: Scheduler Activations
Fonte: [1] – Machado e Maia, 2004. Adaptação.

A maneira de se alcançar um melhor desempenho é evitando-se as mudanças de modos de acesso desnecessárias (usuário-kernel-usuário). Caso uma *thread* utilize um chamado ao sistema que a coloque no estado de espera, não é necessário que o kernel seja ativado, bastando que a própria biblioteca em modo usuário escalone outra *thread*. Isso é possível porque a biblioteca, em modo usuário, e o kernel, comunicam-se e trabalham de forma cooperativa. Cada camada implementa seu escalonamento de forma independente, porém, trocando informações quando necessário.

Indicações

[1] MACHADO, F.B. e MAIA, L.P. ***Arquitetura de Sistemas Operacionais***. 4.ed. LTC, 2007.

[2] SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. ***Fundamentos de Sistemas Operacionais***. 6.ed. LTC, 2004.

[3] TANENBAUM, A.S. ***Sistemas Operacionais Modernos***. 2.ed. Pearson Brasil, 2007.

[4] OLIVEIRA, R.S., CARISSIMI, A.S., TOSCANI, S.S. ***Sistemas Operacionais***. 3.ed. Sagra-Luzzato. 2004.

Atividades

Atividades

1. Compare os pacotes de *threads* em modo usuário e em modo kernel?
2. Qual a vantagem do *scheduler activations* comparado ao pacote híbrido?
3. Dê exemplos do uso de *threads* no desenvolvimento de aplicativos, como editores de textos e planilhas eletrônicas.
4. Como o uso de *threads* pode melhorar o desempenho de aplicações paralelas em ambientes com múltiplos processadores?
5. Quais os benefícios do uso de *threads* em ambientes cliente-servidor?
6. Como o uso de *threads* pode ser útil em arquiteturas microkernel?